# Malybuzz: user's manual

This manual describes Malybuzz and how you can configure it before beginning a fuzzing session. Your configuration will determine which tests will be run and where they will be entered, as well as times and other important parameters. The manual also gives information on the options you have available when a session is being executed, how to interrupt it and how to resume it, among other things. Finally, the manual includes data that you will find useful when adding a new protocol to the application; it gives all the explanations you need so that no doubt will arise when the time comes to make that addition.

# 1. Malybuzz

Malybuzz is the name of this tool, written in Python and devoted to discovering programming errors in network software. This type of application is called a fuzzer, and its function is to create altered requests of the desired protocol with the aim of putting the target software in an unexpected situation that it does not know how to handle. It is a known fact that the creators of today's network protocol-based software do not take enough security precautions, and this failing can be an excellent entryway into a system, or at least a way of causing a denial of the service in question.

A normal execution of the tool goes through several steps. First the altered commands are generated according to the configurations, depending on the application to be tested and the protocol used. Later the communication channel with the target is established, and the fuzzer starts sending requests, which are nothing more than the previously generated commands. For each command sent, a response will be awaited, and a check will be made to see whether the response is correct according to the specification of the protocol in question. If it is incorrect, the error will be notified; if it is correct, the fuzzer will go on sending the remaining commands until it reaches the end of the list.

This tool does not need a specific installation; it is sufficient to decompress the *Malybuzz.tar.gz* file in the directory you choose and execute it. The structure that is created after the decompression is made up of ten directories—*Communication*, *Core*, *Fuzzing*, *Logs*, *Peach*, *ProtocolParser*, *Protocols*, *Restore*, *Tmp* and *Utils*—and a start script. The content of these directories is outlined below:

- *Malybuzz.py*: this is the application start script from which the tests are launched. It has diverse configuration parameters, as we will explain below. It most basic form of execution entails providing a target host, a transport protocol (TCP or UDP), and an application-level protocol (for the time being, FTP or SIP).

- *Communication*: this directory holds the implementation of the tool's communication module logic, which is responsible for sending data to and receiving it from the network.

- *Core*: the Core includes the protocol control programming; it is one of the most important components of the application.

- *Fuzzing*: through the module included in this folder, the diverse alterations of the data can be created, according to the type of fuzzing chosen. It can be said that most of the command creation function is found here.

- *Logs*: if the tool has been configured to do so, it will be here that the session information will be stored, according to the level of detail specified.

- *Peach*: this folder contains an open-code fuzzing library created and maintained to date by Michael Eddington. It provides functions related with the generation of malformed data, management of protocols and communications, among other things.

- *ProtocolParser*: the implementation of the protocol parser is found here. The protocol parser is charged with converting the data provided in the XML specification files to structures that are valid in Python.

- *Protocols*: as its name indicates, this directory contains the specifications of each protocol. It is divided into two folders, one of which is devoted to the specific functions of each protocol—*Functions*—while the other contains the specifications themselves—*Specifications*. The latter directory will be subdivided according to each transport protocol and each specific application protocol. Each *X* application protocol folder will contain two XML files: one with the specification and the possible commands, *X.xml*, and another with the responses to those commands, *X_states.xml*.

- *Restore*: this folder will store both the saved fuzzing sessions and the commands generated during the execution of the tool. The commands will have their own subdirectory entitled *Commands*.

- *Tmp*: as its name indicates, this is the place where the temporary files generated during the execution of the tool will be kept.

- *Utils*: the implementation of methods and classes that are not directly related with the tool logic are found in this directory.

As we have indicated previously, the most basic execution of the tool will be accomplished by launching the *Malybuzz.py* start script and giving it, as parameters, the IP or the name of the host you want to check, the transport layer protocol used, and the application protocol used by the target software. The following is an example of this process:

```
Malybuzz.py 192.168.0.10 tcp ftp
```

*Figure 1. Basic application start command*

After a brief description of the application, the following sections will give greater detail of the ins and outs of this tool.

## 2. Configuration

With a tool of this type, a good configuration is critically important: most of the success of the application will depend on your configuration. This is because the check is not always executed against the same type of software; the applications to be tested can be very different, and each one will require different parameters. This section is divided into two subsections. The first deals with the configuration of protocol fuzzing, showing you how to create new configuration files depending on your objectives. The second focuses on different options that can be passed to the start script at the time of execution. Both are important, and due to the great heterogeneity of the different applications that any given protocol will have, a brief period should be set aside for learning the behavior of the target application, in order to ensure the best results.

### 2.1. Fuzzing

Before we start to explain what the possibilities are in this area, it is important to emphasize that the protocol base files must never be modified; rather, they must be copied with another name and, when you have made the copy, you may make the relevant changes if you wish. These files, as we have indicated above, are found in the directory that bears the protocol name. One will take charge of specifying the commands that can be launched in this protocol, while the other will take care of the possible responses to each of those commands. Because there may be more than two files in these directories, the files bearing the protocol name will be defined as base files, using the following notation: if, for example, the protocol in question is FTP, the commands file will be named *ftp.xml*, while the response file will be *ftp_states.xml*.

*2.1.1. Commands file*

This is an XML file that indicates the specification governing the requests that can be sent for a given protocol. The structure of this file is broken down as follows:

- *port*: application default port. We recommend leaving it intact and, if necessary, modifying this parameter when launching the start script.

- *end-line* and *end-command*: enable you to indicate the format of the request end-lines. "CR" stands for carriage return, "LR" is line return, and "CRLF" indicates both together.

- *com*: this is the label to enter each one of the possible protocol commands. It has a highly important attribute: the *id* command identifier, which serves to distinguish among the various commands. This element contains a sub-element that indicates the name of the command, *com-name*, and another, optional one, *arg*, of which there can be zero or more, and which specifies the possible command arguments. The values of these two labels will be different for each command.

- *globa-arg*: this has the same mission as *arg*, but with a fundamental difference: these elements are common to all of the commands, unless you establish the contrary.

- *field*: this label serves to enumerate the additional protocol fields and their value, *field-value*; this can be broken down in turn into the specific literal *value*, and its possible parameters, *parameter*. In the same way, these parameters are divided into name, *param-name*, and value, *param-value*. The parameter names are separated from the other fields by an initial character, *beginChar*, and from their value by a final character: *endChar*.

- *body*: finally there is the body of the request, which can be completely dispensable.

Now that you are acquainted with the skeleton, we can go on to detail the way to configure it through the various attributes:

- *send*: allows you to decide, through the "yes" or "no" values, whether you want to send the command. As you can imagine, this enables you to fuzz one command without having to send all the rest. Normally, you will only include the commands with which you wish to make the test; however, the case may arise where a command needs another to reach that point in the protocol. For example, in the FTP protocol, even if you only want to check the *PASS*, command, you will need the *USER* command in order to reach the point where the password is requested. In this case both should have the value "yes" in the *send* attribute. The same is true of arguments, fields and parameters; the following elements are affected by this attribute: *com*, *global-arg*, *field*, *parameter* and *body.*

- *command*: together with the previous one, this parameter is valid when you assign the "yes" value to *send* and indicate with which commands the element in question is going to be sent. Its possible values will be the various *id*s separated by commas. For example, in Figure 2 the user is indicating that the *Accept* field will be added only when commands 3, 4, 5 and 6 are sent. This configuration can be set up in the *global-arg*, *field*, *parameter* and *body* nodes.

```
<field send="yes" command="3,4,5,6">
      <field-name type="text">Accept:</field-name>
      <field-value>
            <value type="text">text/*,</value>
      </field-value>
</field>
```

*Figure 2. Example of the command attribute*

- *fuzzing*: with this option you can detail the type of fuzzing you wish to perform in the value of a given node. More specifically, these nodes may be *com-name*, *arg*, *global-arg*, *field-name*, *value*, *param-name*, *param-value* and *body*. The possible values, which can be separated by commas, are the following:

    - *overflow*: this value is based on the repetition of a specific character in search of errors of the *buffer overflow* type.

    - *formatString*: a list of values is used that may lead an application to behave in an unexpected way when certain format strings are entered. These values are also repeated, but less than *overflow*.

- *badString*: the default value is replaced by the elements of a list of different text strings, including uncommon characters, line feeds, etc.

- *badNumber*: with this option, errors of the *integer overflow* type are sought.

- *badPath*: thanks to this value, route handling errors in the tested applications can be found.

- *badIp, badHost*: with *badIP*, malformed IPs are sought, and *badHost* looks for malformed host names.

- *repeat*: this technique enables the default value to be repeated, with the intention of creating an anomalous situation.

- *sql*: this is fed from a list of common SQL injection errors to replace the default values.

- *binary*: characters are generated in different binary mode representations to confuse the target application.

- *fuzzCommand*: this attribute is quite similar to *command*, except that, in this case, you will need to specify in which commands the previously described fuzzing is going to be done. It is valid for the *arg*, *global-arg*, *field-name*, *value*, *param-name*, *param-value* and *body* elements.

Finally, the default values of terminal nodes, that is, of the *com-name*, *arg, global-arg*, *field-name*, *value*, *param-name*, *param-value* and *body* elements, must be established. These values will be the ones that remain when the fuzzing on them has been completed; at this point you can go on to fuzz other commands. Special attention must be devoted to the separator characters between commands and arguments, or between fields, for example, since they must be specified at the end of the value. That is, if in the FTP protocol the *USER* command is separated from the user's name by a space, this could be specified as follows (note the space between *USER* and the end of the *com-name* label):

```
<com send="yes" id="1">
     <com-name type="text">USER </com-name>
     <arg type="text">anonymous</arg>
</com>
```

*Figure 3. Specification of separator characters*

The task of changing the remote host IP in several different places can be tedious; to expedite this sort of work, three different kinds of variables can be used, all of them beginning and ending with the dollar sign character.

- **Element**: begins with the '$X:' sequence, and makes reference to the value of an element of the file. The elements you wish to find by this method must contain a *name* attribute which is unique throughout the file. For example, to make reference to the body of the request, the *$X:BODY$* variable is used, and the body should contain the *name="BODY"* attribute.

- **Variable**: must begin with the '$V:' characters, which serve to obtain, in its place, a value of the static variable specified and stored in the internal implementation of the fuzzer. In this way you can access the values of the following variables:

    - *$V:REMOTE_ADDRESS$*: remote address of the target host.
    - *$V:REMOTE_PORT$*: remote application port.
    - *$V:LOCAL_ADDRESS$*: local address of the machine.
    - *$V:LOCAL_PORT$*: local address used by the fuzzer.
    - *$V:TRANSPORT_PROTOCOL$*: transport protocol used.
    - *$V:APP_PROTOCOL$*: application protocol.
    - *$V:ACTUAL_COMMAND$*: command currently being parsed.

```
<field send="yes" command="1,2,3,4,5,6,9" comment="Mandatory">
 <field-name type="text">CSeq:</field-name>
 <field-value>
       <value type="integer" range="0-2147483648" fuzzing="overflow">4321</value>
 </field-value>
 <field-value>
       <value type="text" fuzzing="badString">$V:ACTUAL_COMMAND$</value>
 </field-value>
</field>
```

*Figure 4. Example of the use of variables in the commands file*

- **Function**: called by writing '$F:' followed by the function you wish to call. Currently only one method is implemented for obtaining the size of a field, but more can easily be added. Thus, to obtain, for example, the size of the body of the request, you would key in the following: *$F:SIZE($X:BODY$) $*.

Another matter to take into account, and a highly important one, is the order of the commands. If you do not specify to the contrary, the fuzzing will begin in sequential order from the first to the last. This is why changing the established order can cause disruptions in the protocol. The following section further discusses the order in which commands are sent.

### *2.1.2. Response file*

The response file describes the possible responses to the commands within a given protocol. Its structure is a lot simpler than the previous one, and can be described as follows:

- *state*: there is one of these nodes for each command. It has an obligatory attribute called *id*, which, as in the case of the *com* element of the commands file, unequivocally identifies the commands. In addition, its value must be included among the identifiers specified in the other configuration, thus establishing a command-response relationship. Its daughter nodes are *name*, which indicates the name of the command, and the two following elements: *response-ok* and *response-err*.

- *response-ok* and *response-err*: each one of these elements indicates a possible response for the specified command. The only difference between the two is that the first refers to positive responses by the application, while the second indicates an error or a waiting state. You can have as many of these parameters as you wish; they may even not exist for a command, which would be a way of indicating that no response should be expected.

Unlike the commands file, in this case the order of the various commands has no consequence when parsing the protocol. The configuration is based on three attributes of the elements that define the responses. Those attributes are:

- *last*: this is an obligatory attribute that indicates, after the response in which it is included, whether a new response is to be expected. Its possible values are "yes" or "no". You will need to modify its value if you find that the behavior of the target application does not respond to what is specified for this field. For example, when reception is supposed to cease after a code *200* from an FTP server, but there is a particular case in which, after *200*, a *260* is sent. In this case, you would have to change the value of this attribute to "no".

- *next*: this option is used to define the next command that must be sent after receiving the response that contains it.

```
<response-err last="yes" next="CANCEL">502</response-err>
```

*Figure 5. Example of the use of the next attribute*

This line, for example, indicates that, after a code *502* is received, a *CANCEL* command will be sent. Therefore, the possible values must coincide with the ones included in the *com-name* elements of the commands file.

- *action*: in this optional field, you may specify the action that must be executed after receiving the response indicated above. Its possible values are subject to the implementation of each protocol; you can examine them from the protocol file located in the *Protocols/Functions* folder.

```
<response-err name="Busy Here" last="yes" action="wait">486</response-err>
```

*Figure 6. Example of the use of the action attribute*

In this example of the SIP protocol, it is indicated that the *wait* method must be executed when a response *486* is received, which will cause the fuzzer to wait a certain time before continuing with the session.

It is also important to stress that, because a target application may not be entirely compliant with standards, the response received may not be the expected one, and the fuzzer may indicate this through the *"Answer not expected"* message. In this case you are free to create a new file and add this response to the specific command.

As you have seen, the configuration in this case is quite simple, but its ease doesn't make it any less important, because the command jumps in the protocol, for example, may be crucial when finding an instance of vulnerability.

### 2.2. Start script options

When the application is executed, certain obligatory parameters must be passed, as well as several optional ones that are also important in the fuzzing result, above all with respect to response times and sending commands.

```
Usage: Malybuzz.py <target_host> <'tcp'|'udp'> <protocol> [options]
Usage: Malybuzz.py <target_host> <'tcp'|'udp'> <protocol> -g <commands_xml_file> <output_file>
Usage: Malybuzz.py -z <fuzzing_session_file> [options]
Usage: Malybuzz.py -h

 OPTIONS

 -h : print this help.
 -v : set the verbose mode.
 -k : keep the fuzzing state when session is aborted,so it will be possible to resume it. The
        session file is stored in the Restore folder.
 -b : set it if the protocol to fuzz returns a banner.
 -p <remote_port> : set the remote port.
 -s <source_port> : set the local port.
 -t1 <response_timeout> : set the response timeout in milliseconds after being sent a request.
                       By default 100.000 ms. If 0 no timeout.
 -t2 <ping_timeout> : set the timeout in milliseconds after being sent a ping probe.
                       By default 2 ms.
 -t3 <between_time_requests> : set the time in milliseconds between fuzzing requests.
                              By default 500 ms.
 -c <max_timeouts> : continue on timeout and set the maximum number of timeouts. If 0 no limit.
 -f <first_command> : set the first command to send in the specified protocol.
 -xc <xml_file> : set the commands configuration file we want to use to do the fuzzing.
                No sense with -z.
 -xr <xml_file> : set the responses configuration file we want to use to do the fuzzing.
                No sense with -z.
 -o <commands_file> : open a commands file and load it in the actual fuzzing session. It must be in
                       the 'Restore/Commands' folder. No sense with -z.
 -l 1|2|3 : log the fuzzing session in the 'Logs' folder with the specified level.
 1: Critical,error and warning messages.
 2: Level 1 plus info messages.
 3: Level 2 plus debug messages.
 -z <fuzzing_session_file> : resume a fuzzing session from a session file.
 -g <xml_file> <output_file> : generate the commands we want to send extracted from a xml file and
                              save them in the Restore folder. No sense with -z.
```

*Figure 7. Application help*

The figure above shows the application help, where each of the possible options is specified. Apart from printing the help texts, there are three ways to launch the fuzzer:

- **Basic**: this is the way of executing the application that we have already described. Its launch is accompanied by the remote host address parameters, the transport protocol used, the application level protocol, and the desired options.

```
Usage: Malybuzz.py <target_host> <'tcp'|'udp'> <protocol> [options]
```

*Figure 8. Use of the basic execution mode*

In this mode the previously generated commands will be generated or loaded, and they will be sent to the target application.

- **Generation**: with this procedure you will generate the commands and their subsequent storage in the *Restore/Commands* directory for their use with different applications—for example, avoiding the continuous generation of the commands with the resulting waste of time. Also, the same obligatory parameters are passed to it as to the previous one, and a single *–g* option that indicates the choice of mode.

```
Usage: Malybuzz.py <target_host> <'tcp'|'udp'> <protocol> -g <commands_xml_file> <output_file> [options]
```

*Figure 9. Use of the command generation mode*

Then you need to provide the XML file route for specifying the protocol commands, as well as a prefix to identify the commands that are to be generated, and which will be used to name the resulting files.

- **Resumption**: with the *–z* option followed by the desired session file route, you can recover a session that, for whatever reason, was not able to be completed. The obligatory parameters of the two previous modes are not necessary; also, you may specify different options at the end. On the other hand, in this type of execution, it would not make sense to specify XML configuration files, nor the *–g* option.

```
Usage: Malybuzz.py -z <fuzzing_session_file> [options]
```

*Figure 10. Use of session recovery mode*

This mode will take care of loading all the options currently in force when the fuzzing was executed during the specified session, including the commands at the exact stopping point, and the protocol configuration files.

Now that you have seen the different ways of executing the script, the various options available can be detailed:

- *-v*: allows you to activate the *verbose* mode, which provides a greater quantity of information per screen during the session. For example, the various commands sent will be printed, along with the responses.

- *-k*: using this option will enable you to save the session if some incident occurs during its processing, or if you wish to abort the execution by using the *Ctrl+C* keyboard combination. This will allow you to return at a convenient time to the exact point where the fuzzing session was interrupted.

- *-b*: this is a function conceived for connection-oriented protocols, based on the TCP protocol, and which enables you to specify whether, at the time of connection with the remote host, a *banner* or welcome message must be awaited from the target application.

- *-p <remote_port>*: the remote port is specified where the application you want to check is to be listened to.

- *-s <source_port>*: allows you to choose the local port to which the commands will be sent, and where responses will be awaited. If you do not have superuser permissions, you must specify a port larger than 1024, since these are reserved, and their use is not recommended.

- *-t1 <response_timeout>*: specifies the maximum time to wait for a response from the fuzzed application. The time must be entered in milliseconds, and is set by default at 100,000 ms. If the figure you enter is 0, it will mean that no wait time is set, and thus that the waiting time is unlimited.

- *-t2 <ping_timeout>*: before each command send, a little test is run to check whether the remote application responds correctly. The maximum response time in milliseconds is set with this option; the default value is 2 ms.

- *-t3 <between_requests_time>*: this option is used to establish the time that must elapse between the sending of a command and the sending of the next one. You will need to enter the value in milliseconds; the default value is 500 ms.

- *-c <max_timeouts>*: if the time you specified under the *–t1* option is exceeded, the value you assign to this option will decide the number of *timeouts* that must be awaited before aborting the fuzzing. If you enter a value of 0, you are indicating that there is no limit for *timeouts*.

- *-f <first_command>*: thanks to this parameter, you can choose the command you wish to use to start the fuzzing process. This may be useful, for example, when the protocol status has become stuck on a specific command and you wish to get out of this situation. In this case, you would abort the session and restart by using this option.

- *-xc <xml_file>*: indicates the XML file route for specifying the protocol commands that will be used in the session currently under way. The base file is used by default.

- *-xr <xml_file>*: indicates the XML file route for specifying the protocol responses that will be used in the session currently under way. The base file is used by default.

- *-o <commands_file>*: this option is used to specify the previously generated commands file that will be used to load the commands for the current session. This will enable you to save time when generating commands, and is useful when you want to apply the same fuzzing process to different applications.

- *-l 1|2|3*: you can use this parameter to indicate that you wish to save a log file of the current session. The number you will need to enter serves to define the degree of detail with which you want the events to be recorded. Entering 1 will produce the simplest level, while 3 will ensure that debugging information is also recorded. This file will be saved in the *Logs* directory.

Similarly, it is important to point out that the great majority of errors associated with timing that cause fuzzing sessions to end prematurely are due to a poor configuration of the various times. For example, if too many successive requests are sent to an application, the most probable occurrence is that the application will reject the connections. The solution for this particular case would be to use option *–t3* to increase the time between requests. When you are faced with this sort of problem, you must adjust the various configurable times to obtain an optimum result.

# 3. Adding a new protocol

The inclusion of a new protocol will be easy or difficult, depending on the protocol and the degree of detail you wish to specify. For the moment, this fuzzer takes care of flat text protocols, which means that their specification may be simpler. Basically, you will need to take two steps: the creation of the XML command and response specification files, and the implementation in Python of the methods required to respond to certain received code. Therefore, you could omit the latter step if you wanted a simple fuzzing of the protocol.

## 3.1. Creating specification files

Before you start to write the files, if you are working with a TCP-based protocol, you must create a folder with the name of the protocol in lower case letters within the *Protocols/Specifications/TCP* directory. If your protocol is based on UDP, the directory will be *Protocols/Specifications/UDP*. Once you have accomplished this, you will need to create two files called *protocol.xml* and *protocol_states.xml*, where *protocol* is the name of the folder containing them and of the protocol to be specified. Since the details of the contents of these files were given in section 2.1 above, we won't describe them here. Below you will see the structure and the relationships among the various elements of the command and response files, respectively.

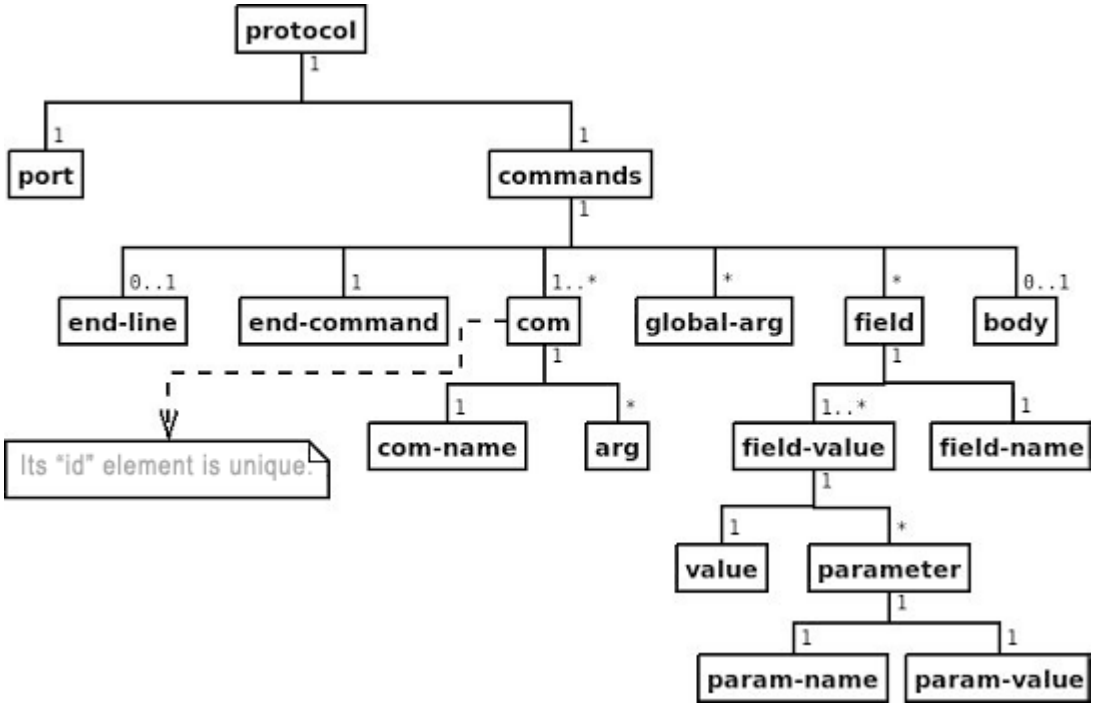


*Figure 11. Structure of the commands file*

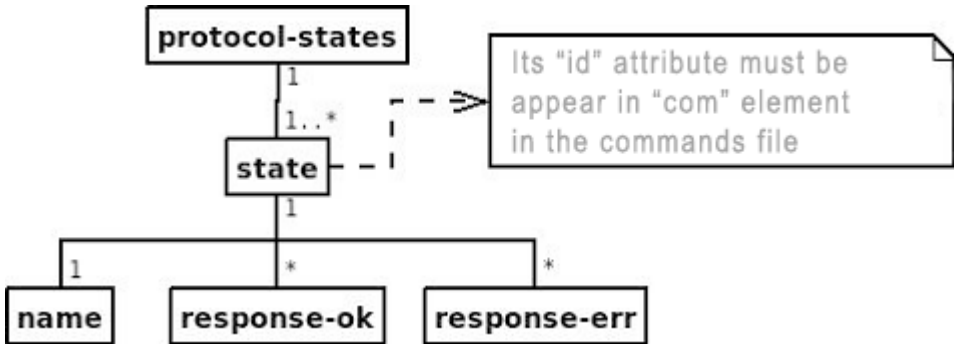


*Figure 12. Structure of the response file*

### 3.2. Protocol implementation

If you wish to execute an action when a certain response is received, through the *action* attribute of the response file, you will have to implement a method bearing the same name in the relevant Python module. That is, you must create a module called *protocol.py*, where *protocol* is the name of the protocol, in the *Protocols/Functions* directory. It is in that module where you must implement the method with the unique and obligatory parameters that are detailed below:

```
def wait(command = "", response = "", logMode = 0, verboseMode = 0, endLine = '\n', xmlFiles = []):
        time.sleep(5)
```

*Figure 13. Example of a specific protocol method*

- *command*: command sent to the target application. By default, it is the empty string.
- *response*: response from the application to the command sent.
- *logMode*: indicates whether the events must be registered in a file. This is deactivated by default.
- *verboseMode*: indicates whether details must be given on screen during the fuzzing session. This is deactivated by default.
- *endLine*: specifies the end-line used in commands and responses. By default, it is the line return.

- *xmlFiles:* this is a tuple comprised by the configuration files used in the session.

When you have done this, it will be sufficient to include the appropriate value in the *action* attribute to finish the development. In this example, this would be *action = "wait"*.